

Design and Implementation of the Multiuser Index-based Data Access System*

Pyung-Chul Kim, Hwan-Ik Choi, Yoon-Joon Lee**

Computer Science Department
Korea Advanced Institute of Science and Technology
P.O. Box 150, Chungryang, Dongdaemun, Seoul, 130-650, Korea

Myung-Joon Kim

Database Section
Electronics and Telecommunications Research Institute
P.O. Box 8, Daeduk Science Town, Taejeon, 305-606, Korea

Abstract

We have built a multiuser storage system named MIDAS (Multiuser Index-based Data Access System) which is intended to be used for a lower subsystem of database management systems running on UNIX. MIDAS implements a modified B⁺-tree structure for the sake of control of concurrent operations on a B⁺-tree index. To guarantee the more concurrent accesses to the system tables such as buffers, lock table, etc., MIDAS divides each system table into several independent partitions. A MIDAS database can have more than one disk volume and a file can be spread over several volumes to deal with very large files and databases. We overcame the flaws of the UNIX file system efficiently by implementing an extent-based disk manager by means of the raw device interface. We employed the two phase locking method in conjunction with multiple granularity to serialize executions of several transactions which access to shared databases. This paper addresses the design rationales incorporated within the implementation, system architecture and results of the benchmark which was carried out in order to verify the design and implementation decisions.

1 Introduction

The database technology is very useful to deal with very large volume of data in many applications such as on-line transaction processing systems, file management systems, mailing systems, etc. With the help of many research communities, the conventional database technology has been matured. At this point, there are many commercial relational database management systems (DBMSs) available: Ingres from Relational Technology, Inc., Database 2 from IBM, Informix-Turbo from Informix Software, Inc. and so on.

* This work was in part supported by Electronics and Telecommunications Research Institute under the Contract number M89-014.

** To whom correspondences should be addressed (e-mail: yjlee@sorak.kaist.ac.kr).

A DBMS has a data access system* as its lower subsystem. A data access system manages various storage structures such as sequential files and indices to be used for efficient key-associative accesses to large volume of data and provides methods to access the storage structures for a upper subsystem, i.e., a query processor. Moreover, a data access system has concurrency control and recovery facility to support transaction concept (atomic action).

RSS (Relational Storage System) which is the data access system of System R [Astr76] is generally accepted as one of pioneers. RSS revealed the techniques for implementations of various storage structures including sequential files and B⁺-tree indices, a concurrency control method and recovery facility from failures.

WiSS (Wisconsin Storage System) is a single user storage system which was developed in University of Wisconsin at Madison [Chou85a]. WiSS makes use of raw device interface of UNIX to get high performance and provides B⁺-tree, extendible hash as indices and supports a storage structure to store arbitrarily long data items. At this point, the multiuser version of WiSS called MWiSS which was made as part of the Gamma relational database machine project is available [Ghan89].

The lower subsystem of Informix-Turbo also runs in place of the UNIX file system by use of raw device interface and manages system tables using the shared memory facility of UNIX System V interprocess communication (IPC) package [Info87]. The lower subsystem of Informix-Turbo is called RSAM (Relational Storage Access Method). RSAM manages sequential files and B⁺-tree indices.

To build our own storage system, we had developed a prototype of a multiuser storage system called MUSE [Kim89] by rewriting and adding some modules including a lock manager to WiSS. With the help of experiences obtained in the MUSE project, We have built a multiuser storage system named MIDAS (Multiuser Index-based Data Access System) which is intended to be used for a lower subsystem of DBMSs running on UNIX. This paper addresses the goals incorporated within the implementation, system architecture and results of the benchmark which was carried out in order to verify the design and implementation decisions.

2 Implementation Goals

In the NAIS (National Administration Information System) project which is funded by the government, we need a DBMS to deal with very large databases efficiently. The purpose of our project is to develop a high performance multiuser storage system running on top of UNIX operating systems so that it can be used as a lower subsystem of relational DBMSs in the NAIS project.

* The terminologies, "data access system", "storage system" and "lower subsystem" are used interchangeably with the same meaning in this paper.

So far, many storage systems and DBMSs with their own storage systems have been developed by vendors, universities or research institutes. But they are not available in public domain or their implementation philosophies are not suitable for our purpose. The goal of our project is not to implement another feature but to build our own storage system which is suitable for the NAIS environment.

The storage system is run on general purpose operating systems such as UNIX and its variations because it becomes a quasi standard. By use of general purpose operating systems we can obtain several benefits such as portability of the developed storage system, easiness of application development with the help of various tools and reusability of application programs.

Tightly coupled multiprocessor computers will be used in the NAIS environment. Therefore, we make efforts to obtain more parallelism and concurrency. A data structure is designed to increase concurrency of accesses to hot spot system tables. Original B+-tree structure is modified to overcome the limitation of concurrency inherited from the tree structure. In the current version of MIDAS, however, there are few achievements for developing new parallel algorithms such as parallel sort and merge.

As many researchers have pointed out, there are many pitfalls in a storage system which runs on top of and not in place of the UNIX file system [Ston81, Andr89]. To guarantee very high performance in UNIX, we circumvent the UNIX file system efficiently by implementing an extent-based disk manager using the raw device interface.

In order for MIDAS to be used in the NAIS applications, it should be able to manipulate very large volume of data. The size of a file is expected to be reached up to tens of gigabytes. To do this, a file can be spread over several disk volumes and a database can comprise more than one disk.

MIDAS should provide transaction processing facilities such as concurrency control to serialize the interleaved executions of transactions and recovery to keep databases consistent from various failures.

Our storage system is implemented to be easily migrated from one UNIX-based machine to another. For example, we reduce assembly codes as possible, declare variables as they do not cause overflows in two or four byte machines and define data structures as aligned to be independent of a specific compiler.

3 System Structure

The MIDAS system consists of a library of functions with which a DBMS can be built by linking and some utility commands including initialization and finalization of the MIDAS system and getting statistic of databases. MIDAS is written in C programming language to get efficiency and productivity in the UNIX environment. As shown in Figure 1, MIDAS consists of twelve distinct modules and they are grouped into four categories according to their roles: communication, transaction, storage and shared memory management.

The modules in communication management category manage application processes (client manager), DBMS processes (server manager) and queue for messages between these two kinds of processes (queue manager). The feature of this category is highly dependent on the relationship between application and DBMS processes. Three of the relationships: only one DBMS process for all application processes, a DBMS process for each application process and fixed number of DBMS processes for all application processes, are generally considered [Hærd84]. Currently, MIDAS assumes the relationship where a DBMS process is assigned to each application process because it is very simple to implement. However, we may change it after the investigation into

which relationship is the most preferable in database applications. For this reason, we do not describe the details of communication management category in this paper.

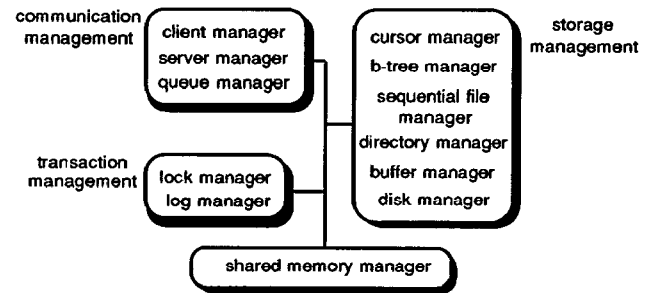


Figure 1. The MIDAS system structure

The modules in transaction management category support concurrency control (lock manager) and recovery facility (log manager) for transaction concept. We employed the strict two-phase locking in conjunction with multiple granularity [Gray78] for concurrency control and write ahead log protocol [Pete83] for recovery.

The purpose of storage management category is to map user objects such as records and files into disk objects such as disk volumes, extents and pages. The functions of the modules in this category are as follows: cursor manager provides navigational accesses using the cursor concept which is an abstraction of current record qualified by given predicates, B+-tree manager manages B+-tree indices for efficient key-associative accesses to large data files, sequential file manager handles data files which are doubly linked list of pages, directory manager maintains an access method to find the descriptor of a file efficiently, buffer manager keeps highly used pages of disks in main memory to reduce disk inputs and outputs and the disk manager organizes physical disk volumes as the MIDAS database structure.

The system tables such as open file table, buffers, etc., should be accessible by more than one DBMS process and they must be kept consistent from the interleaved accesses of DBMS processes. Shared memory manager allocates shared memory segments for MIDAS system tables and provides mutual exclusion primitives.

3.1 Shared memory management

The system tables such as open file table, buffers, etc., should be visible to several DBMS processes. A region of memory can be shared by two or more independent processes by use of the shared memory facility of the UNIX System V IPC package. At initialization of the MIDAS system, MIDAS allocates shared memory segments from the operating system and initializes them to be used for system tables. Thereby, each DBMS process attaches the system tables at its address space and then accesses them.

Since processes are scheduled preemptively by the UNIX operating system, accesses to the system tables can be interrupted at arbitrary times. Therefore, system tables should be accessed in mutually exclusive manner to keep them consistent. We rule out direct use of the system semaphore facility of UNIX System V IPC package for performance reason. Instead, we implement two versions of mutual exclusion primitives: a blocked-wait locking and a busy-wait locking.

For the blocked-wait locking a conditional semaphore facility is implemented using a TS (Test and Set) machine instruction and the system semaphore facility. If a system table is not locked, a DBMS process can lock it just only by a few machine instructions including a TS instruction, otherwise, the process waits by use of the system semaphore until the table is unlocked. The conditional semaphore

facility can reduce the overhead of the system semaphore when the probability is high that system tables are not locked.

In the busy-wait locking a DBMS process executes a TS instruction repetitively to acquire a lock. In the multiprocessor case a busy-wait locking is expected to be more efficient because it is likely that a process which holds the lock runs on another processor and releases it soon [Blas79].

However, both of the locking methods issue non-zero probability of starvation of a process because a lock is granted to the process which gets a CPU first by racing right after the lock is released. This problem can be solved by keeping a line of waiting processes and scheduling them in FIFO manner. But, we do not solve this problem because the probability is expected to be very low and the solution is complex to implement.

3.2 Disk management

As many researchers have pointed out, the UNIX file system is although very flexible but has many shortcomings in terms of database supports [Ston81, Andr89]. The UNIX file system is organized as relatively small pages and does not cluster the pages which are highly relevant, which causes significant amount of time to seek for desired pages. Reading or writing a disk page has additional overhead of copying the page from or to UNIX buffers. Writing an updated page is not synchronized to disks due to the delayed write policy of the UNIX buffer management, which causes a serious problem in the implementation of recovery facility. The UNIX file system also needs additional searches for indirections to locate the current position in a relatively large file.

In order to avoid these pitfalls, MIDAS runs in place of the UNIX file system by use of the raw device interface. Since page inputs and outputs of a raw device bypass the physical address translation and buffering of the UNIX file system, the shortcomings described above can be circumvented.

A MIDAS database consists of several raw disk volumes and each volume is divided into equi-size extents which are physically contiguous groups of pages. The size of a page is 4096 bytes. A file is represented as a chain of extents which may belong to different volumes and an extent cannot be shared by more than one file. The size of an extent in each volume is given at the creation of a database and should be calculated by weighing performance and fragments. The larger size of extents is expected to obtain more performance by reducing seek times but issues more space wastes due to fragmentations of extents.

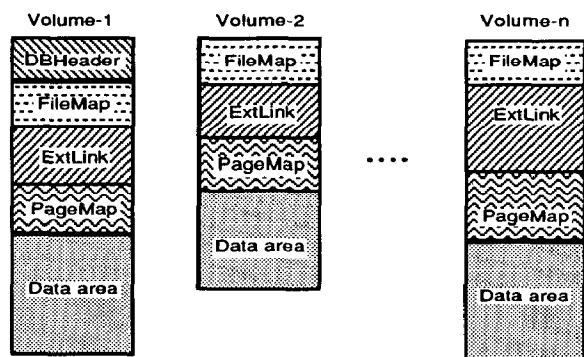


Figure 2. MIDAS Database Organization

Figure 2 shows the MIDAS database organization. There is a control header (DBHeader) located at the first page of the first volume of each database. DBHeader contains descriptions such as names,

extent sizes, numbers of free extents and so forth, of each volume which belongs to the database. Each volume has three kinds of pages at the beginning: a file map page (FileMap), extent link pages (ExtLink) and page map pages (PageMap). FileMap and ExtLink are to identify the extents allocated for each file. In order to speed up allocation of a page, the usable extent which contains free pages is maintained for each file. PageMap specifies whether a page is allocated or free.

A file can be spread over several volumes in order to store very large files. Extents for a file are allocated starting with the n -th volume, where n is the remainder when the file number is divided by number of volumes so that disk operations on different files can be parallelized if possible.

A database should be mounted first to be accessed. At mounting a database MIDAS reads the pages of DBHeader, FileMap, ExtLink and PageMap in the database into the database table in shared memory to avoid the overhead of frequent disk accesses to the pages. All updates in these pages are flushed out to disks at dismounting time.

3.3 Buffer management

The role of buffer management is to provide pages cached in main memory for upper modules. It calls the disk manager to read or write pages from or to disks, maintains a buffer pool of pages and implements an efficient page replacement policy to reduce disk inputs and outputs.

Since Stonebraker [Ston81] has observed that the LRU replacement strategy appears to perform only marginally in database environments, there have been researches proposing new buffer management algorithms to obtain more hit-ratio by utilizing prediction of access patterns of pages [Chou85b, Sacc86, Jauh90]. They are, however, complex to implement and their performance characteristics are not analyzed deeply in multiuser environments. At this point, we simply implemented the LRU replacement strategy.

A buffer table is one of high traffic resources in database systems. Since the UNIX operating system has a preemptive scheduler, high traffic locks on a buffer table may cause a convoy phenomenon [Blas79]. MIDAS divides the buffer table into several independent partitions to scatter the heavy traffic locks. As shown in Figure 3, each partition has an independent hash table and an LRU chain. A chained-bucket hash is implemented as a technique to treat overflows in the hash tables and a doubly-linked list is used to keep the LRU chains. In order for the MIDAS buffer manager to find a page, it first calculates the partition number by applying a hash function to the page identifier and then computes the home bucket address of the partition by hash again and

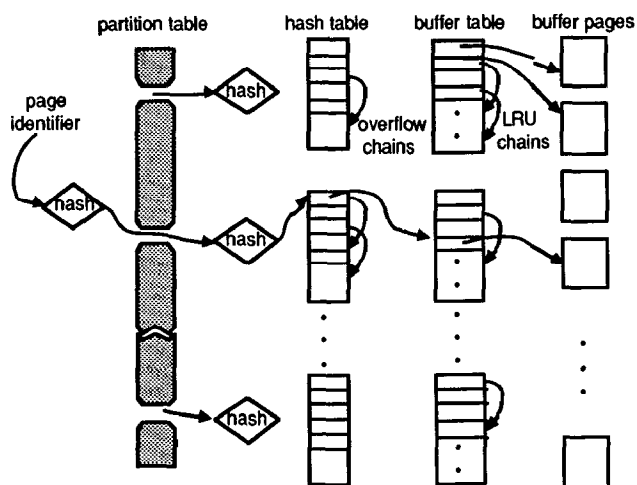


Figure 3. Structure of the MIDAS buffer

then searches the page through overflow chains. While a partition is being accessed, hash tables and LRU chains of other partitions can be updated simultaneously.

3.4 Directory management

In MIDAS each database has an internal file directory to associate an external file name with its internal information, i.e., a file descriptor. The information of a file descriptor includes disk level file identifier, the first and the last page of the file, the address of descriptors of indices built on the file, etc. In order to speed up finding the descriptor of a file with an external file name, we organize the directory as an extendible hash structure [Fagi79]. The extendible hash structure which we implemented has only one page for hash table as does in WiSS. Since a page identifier is 4 bytes and a page is 4096 bytes, the global depth can grow up to 10 ($= \log_2(4096/4)$). When a file descriptor is to be inserted to a full filled leaf page whose local depth has reached 10 already, an overflow page is created and chained to the leaf page of primary bucket rather than doubling the hash table to occupy two pages.

3.5 Sequential file management

The purpose of this module is to manage sequential files and provide operations on them. A sequential file is represented as a doubly-linked list of pages for the sake of forward and backward scans. A record is uniquely identified by the location of it, i.e., the record identifier which comprises page number in the database and slot number in the page. The stored record has the form $\langle \text{length}, \text{data} \rangle$. *Length* can be negative to specify the record is deleted. In such case, negative of *length* is the number of bytes of the deleted record. For the sake of simplicity, addition of a record occurs at the last of a file and the space issued by deletions or updates of records is not cleared immediately but compressed during periodic reorganizations.

The files are classified into two types: regular files and temporary files. Either concurrency control or recovery is not applied for a temporary file. When MIDAS is linked together with a query processor, the query processor can create temporary files to store intermediate results during processing relatively complex queries such as joins because intermediate results do not have to be locked or prepared for recovery.

To enhance performance the file descriptor of an open file is kept in the open file table in shared memory until it is closed. The open file table is also divided into several independent partitions to guarantee more concurrency. Each partition is managed as a chained-bucket hash structure and an integer hash value for each open file is stored to speed up matching a file by comparison of integers instead of strings.

3.6 B⁺-tree management

We chose a B⁺-tree structure [Come79] as a primary access method for efficient scans of records in a data file associated with given key values. A key can be a concatenation of more than one field. An index is either clustered or nonclustered. A clustered index is one where the records in the data file are sorted in the same order of the index. A unique number is assigned to each index in a file and it is used to identify the index. In this approach users may suffer from that they should keep the number for each index in their minds, but a query processor built on MIDAS can easily specify the index number because it can be obtained at query optimization time.

The pages of a B⁺-tree are either internal pages including a root page or leaf pages. Each entry of an internal page has the form $\langle \text{key value}, \text{page number} \rangle$. The key values of all entries in the child page specified by *page number* are greater than or equal to *key value*. All internal pages have three page pointers: two are for the previous and the next page at the same level, one is for the leftmost child page. To avoid the overhead of tree traversal for indexed sequential scan, the leaf pages

are doubly linked together. Each entry of a leaf page has a key value and a record identifier to specify location of the data record which has the key value. All leaf pages have three page pointers: two are to implement the doubly-linked list and one is for overflow page chains.

Many researchers have proposed locking schemes which are suitable for B⁺-trees and modified the original B⁺-tree structure so that the more operations on the B⁺-tree can be executed without blocking each other as possible [Baye77, Lehm81, Sagi85, Mond85, deJo90]. They are, however, complex to implement due to their sophisticated operations. In order to maximize the concurrency of accesses to a B⁺-tree and make it easy to implement at the same time, we simplified the original B⁺-tree structure as follows. When an entry is inserted to a leaf page whose space is not enough, MIDAS allocates a new page and moves the second half of entries in the original page to the new page and links the new page to the original page as an overflow instead of propagating the split to upper level. Deletions of entries do not accompany merge of either leaf pages or internal pages even though the entries in two adjacent pages can be stored into one page.

Figure 4a shows an example of a B⁺-tree without overflow pages. It is assumed that the maximum number of entries of a leaf page is 2. If the record whose key value is 37 is inserted, the leaf page is split and an overflow page is introduced (see Figure 4b).

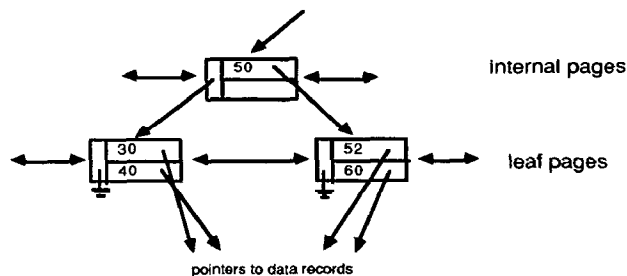


Figure 4a. An example of a B⁺-tree without overflow pages.

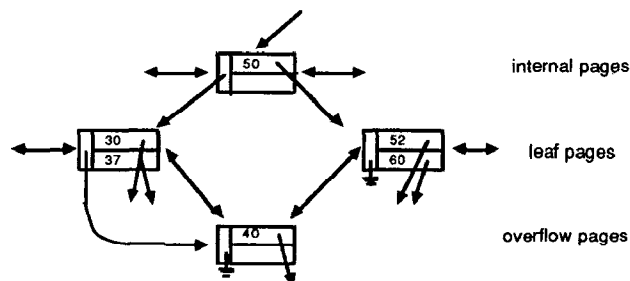


Figure 4b. The B⁺-tree after a new key (37) is inserted.

By such modifications, internal pages of a B⁺-tree are not updated at all once the B⁺-tree is built. Therefore, any internal page does not have to be locked for all operations on a B⁺-tree. The deletion or insertion of an entry in a leaf page may cause movements of other entries because the entries should be kept sorted to utilize binary searches. For this reason, we chose the page granularity as lock level of B⁺-tree indices. MIDAS acquires a lock on the leaf page to be affected before an update and releases the lock after the update.

However, a B⁺-tree in MIDAS can remain unbalanced because there is no merging two adjacent leaf pages and splitting a leaf page is not propagated to internal pages. The unbalance of a B⁺-tree can be disappeared by rebuilding it at the reorganization time. The reorganization of an index can be done just after close of it by a daemon process without blocking read operations on it.

We have implemented a disk-based external 8-way merge-sort with

9 buffers (8 for inputs and one for output). The sort utility can be used for implementation of sort-merge join by a query processor. Since a join is a time-consuming operation, it would be better to reduce the number of records to be involved in the join as possible. In MIDAS a query processor can restrict the records to be sorted by giving a filter as an argument of sort function. When a filter is associated, only qualified records are participated in the sort.

The algorithm to build a MIDAS B⁺-tree is as follows: MIDAS first extracts the pairs <key value, record identifier> from the data file and sorts them. With the help of information on the results of sort and the key descriptor, MIDAS calculates how many pages should be allocated for leaf level and internal levels respectively. For leaf pages the loading factor is assumed 0.9 to reserve spaces for entries to be inserted. The loading factor of internal pages is, however, assumed 1.0 because there can be no modifications. After the calculation, MIDAS allocates the pages and fills entries to each page from leaf pages to a root page straightforward. With this algorithm, we can build an index maximizing storage efficiency and not using the split algorithm caused by repetitive insertions of keys.

3.7 Cursor management

A MIDAS cursor is an abstraction of the current position in the set of qualified records of a data file. A user can move the current location, read or update the record at the current location. For the sake of performance and easy implementation of a query processor, qualifications on various levels can be associated with a cursor and the given qualifications are applied to the records in MIDAS buffers to avoid the overhead of copying them into another temporary area.

The following kinds of qualifications can be given for a cursor associated with an index (see Figure 5): a range of keys to be scanned (KeyRange), a key filter to be applied to key fields of records (KeyFilter), a data filter to be applied to non-key fields of records (DataFilter).

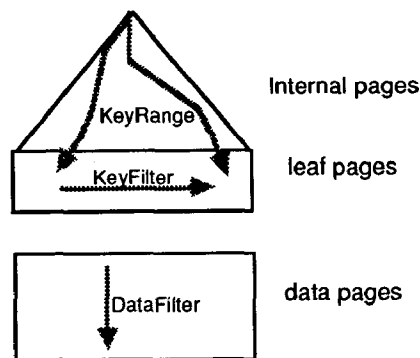


Figure 5. Qualifications on various levels.

KeyRange is specified by two key values for the lower bound and the upper bound with which the cursor starts and ends respectively. A filter is a predicate usable as a search argument which had been proposed by System R [Astr76]. The predicate can be any disjunctive normal form expression where each atomic expression is of the form <field specification, comparison operator, constant value>. When a cursor created, it locates the lower bound and scans entries in leaf pages until the upper bound is reached. During the scan, only records whose keys are evaluated true for KeyFilter are actually searched and then DataFilter is applied for them. After all, the cursor returns only the records satisfying given three qualifications.

To enhance performance of filter applications MIDAS implements the comparison routines for possible data types and comparison

operators by use of a function pointer array rather than switch constructs.

3.8 Lock management

To free a transaction from interferences of others MIDAS serializes accesses to shared data objects such as files and records by a lock-based synchronization mechanism which has been known to be able to obtain high degree of resource utilization [Care84]. MIDAS implements a strict two phase locking method [Bern87] since it is easy to implement. The optimal granularity of locks varies with the characteristics of transactions and system environments [Tay85]. We adopted the multigranularity protocol lest MIDAS should be dependent on specific applications or system environments.

The locking primitives and their compatibility in MIDAS are the same as System R [Gray78]. The granules of locks are files, pages and records. The page granule is used only for concurrency control of B⁺-tree operations. When a cursor is built on a file, the lock modes in file level and record level are determined according to characteristics of the cursor. Figure 6 shows the relationships between lock modes and characteristics of a cursor.

characteristics of a cursor		lock modes	
access method	operation mode	file lock mode	record lock mode
without index	read only	S	implicit S
	read, write	SIX	implicit S and explicit X
	write only	X	explicit X
with an index	read only	IS	explicit S
	read, write	X	explicit S and explicit X
	write only	X	explicit X

(S: Shared, X: Exclusive, IS: Intent Shared, IX: Intent Exclusive, SIX: Shared and Intent Exclusive)

Figure 6. The relationship between lock modes and cursors.

Whenever a transaction starts waiting in line due to lock conflicts, MIDAS checks if the transaction issues a deadlock by keeping a wait-for-graph which represents the wait-for relationship among transactions. When the transaction causes a deadlock, it rolls back and releases all locks which are granted for it. The cycle detection algorithm proposed by Jiang [Jian88] is implemented.

The lock table in MIDAS consists of a file lock table and a record lock table to manage locks on file level and record level respectively. Each lock table is also divided into several independent partitions to obtain more concurrency of accesses. A chained bucket hash structure is used to speed up finding an entry in a partition.

3.9 Log management

The recovery facility is not supported in MIDAS at this point and is expected to be included in a near future. That facility has been already designed and is under implementation. We adopted the write ahead log protocol [Pete83] because it is widely used. In MIDAS, all dirty pages in buffer pool are synchronized to disks at checkpoint to simplify undo scenario. And the group commit protocol [Bern87] or the parallel logging [Agra83] will be employed to overcome the limitation of disk speed.

4 Performance Test

To find out the performance of MIDAS we compared the performance of MIDAS with Informix-Turbo version 1.10.03B and WiSS version 3.0 by use of the Wisconsin benchmark. The purpose of

this comparison is to verify design and implementation rationales which we employed and not to claim that MIDAS outperforms WiSS and Informix-Turbo in all facilities. In this section we describe the benchmark specifications and analyze the results. The detailed descriptions on this benchmark experiment are also available in [Kim90].

4.1 The benchmark configuration

The hardware environment of this benchmark is as follows: a SUN3/50 workstation based on a 2 MIPS 68020 microprocessor, a 4 megabyte main memory and a 140 megabyte Micropolis 1355 disk drive whose average access time is about 20 milliseconds. The disk drive is connected to the SUN3/50 with a Emulex MD21 controller using the ESDI/SCSI interface. In the Wisconsin benchmark, it is recommended that the number of disks should be two: one for the database relations and one for the system including database software to avoid the effect of disk inputs and outputs issued from swapping [Bitt88]. In our benchmark since the main memory of the SUN3/50 workstation is large enough to keep database buffers and test programs without page faults, there is no need to connect another disk for the system software. The disk has 8 heads and 34 sectors per track. The extent size of MIDAS and WiSS is assigned to the size of a cylinder, i.e., 34 pages (34 × 4 kilobytes = 136 kilobytes). The operating system is SunOS Release 3.5.1. The UNIX System V IPC package is turned on for MIDAS and Informix-Turbo to make use of shared memory and semaphore facilities.

The major relational operators such as selection, join and aggregate are implemented as procedures of MIDAS and WiSS so that the results of this benchmark can be an indicator of the performance obtainable by a database management system built upon these systems using a pre-compiling technique. All system tables of WiSS including an open file table and buffers are managed in the data segment which is local to a WiSS test application process.

As the Wisconsin benchmark recommended, 1 megabyte of main memory is dedicated to data buffers for the case of MIDAS and Informix-Turbo. In the WiSS version 3.0, 26 pages are dedicated to sort buffers at the compile time: one is for output buffer and 25 are for input buffers. For the sake of fairness, we assigned the number of data buffers of WiSS to 230 pages (920 kilobytes) instead of 256 pages.

In Informix-Turbo two processes named *sqlturbo* and *isql* are configured for each user. *Sqlturbo* is a back-end process which parses, optimizes and executes SQL queries. *Isql* is a front-end process which submits queries given by the user to *sqlturbo*. When an *isql* process is invoked with a test query, it forks a *sqlturbo* process as its child and sends the query to the child process via pipe. After completion of processing the query from *isql*, *sqlturbo* sends the results to the parent process via pipe. It is expected to be unfair with Informix-Turbo that the benchmark programs for MIDAS and WiSS do not pay the following cost: pipe overhead to send queries, compilation and optimization of queries, validation of access privilege and the overhead to fork a back-end process *sqlturbo*. However, this overhead is taken only once for each test query. From the response time of update queries, this cost is turned out to be less than a second, which is negligible for the case of queries whose response times are larger than a few seconds.

For MIDAS and Informix-Turbo, the shared memory is attached at the logical address, 0x80000 to reduce the swap space required for a test program of MIDAS and a *sqlturbo* process of Informix-Turbo. In the case of MIDAS and Informix-Turbo, the concurrency control is turned off and logging is not applied to the database of Informix-Turbo.

4.2 Test database

In the original Wisconsin benchmark, the test database consists of 5 synthetic relations with identical attributes but different cardinalities

[Bitt83, Bitt88]. In our benchmark, the test database comprises 4 relations: two relations (TenK1 and TenK2) of 10,000 tuples, one relation (BPrime) of 1,000 tuples and one relation (temp) for result storage. Only 2 byte integers are represented in the original Wisconsin benchmark. However, it would be hard to justify 2 byte integers today because most of computers which are workstations or larger use 32 bit processors. In our benchmark, we take 4 byte integers and used 7 integer attributes rather than 13 integer attributes to make the size of a relation comparable to the original one.

For queries with indices, organizations are a primary clustered index on unique2, a nonclustered index on unique1 and a nonclustered index on hundred field. A 30 megabyte raw device partition is reserved for the test database. After the completion of benchmark experiment for MIDAS, the same partition is used for WiSS as well as Informix-Turbo respectively.

4.3 Test queries and results

The elapsed times of the test queries are obtained with the help of *time(1)* command in UNIX. To compute average response times, each query needs to be executed more than once. For selections and updates, 10 instances of a single query are run. For joins and aggregates, 2 instances of a single query are run. Each query is run several times alternating the two 10,000 tuple relations in order to get rid of interferences among queries.

In the original Wisconsin benchmark, there are 32 queries named from Q1 to Q32, which are categorized into five groups: selections, joins, projections, aggregates and updates [Bitt88]. We test only 14 queries of them to verify our storage system. Most queries skipped are either to test the optimization facility of a DBMS rather than the access method of a storage system or not applicable to MIDAS and Informix-Turbo.

Selections

Three cases of query with fixed selectivity 1% are tested: a selection without index (Q1), a selection which takes advantage of a clustered index (Q3) and a selection which only uses a nonclustered index (Q5).

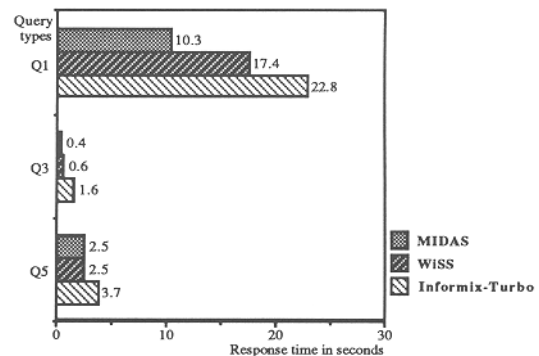


Figure 7. Select queries

For the sequential scan of a data file with search predicates, MIDAS and WiSS are optimized to apply the given predicates to the records in buffer pages instead of copying them into another temporary area. MIDAS implements all comparison routines by means of a function pointer array rather than *switch* constructs of the C language. These features make MIDAS outperform other systems in the selection without an appropriate index (Q1). From the results in Figure 7, we also recognize the performance of an index of MIDAS and WiSS is superior to that of Informix-Turbo (Q3, Q5).

Joins

The join operations in the Wisconsin benchmark are to test the join algorithms and the query optimization. We test the following join queries: joins a ten thousand tuple relation and the results of a selection on another ten thousand tuple relation without indices (Q9), with clustered indices (Q12), with only nonclustered indices (Q15) and joins a ten thousand tuple relation and BPrime without indices (Q10), with clustered indices (Q13), with only nonclustered indices (Q16).

For the join method of MIDAS and WiSS, the nested-iteration join algorithm is applied if the appropriate clustered or nonclustered indices are available, otherwise, the sort-merge join algorithm is applied. The query optimizer of Informix-Turbo always uses the nest-iteration method for join. It builds a temporary nonclustered index if any applicable index does not exist.

The results of joins following a selection are shown in Figure 8a and the results of joins without selections are shown in Figure 8b. In the case of the join query with neither a selection nor indices, Informix-Turbo outperforms other systems (Q10). But, Informix-Turbo falls behind with the processing of join query following a selection (Q9). The fast response time of Q10 in Informix-Turbo is mainly due to the access path selection of query optimizer rather than due to the performance of lower subsystem RSAM. Informix-Turbo is inferior in dealing with the join queries with appropriate indices which is either clustered or nonclustered (Q12, Q13, Q15, Q16).

The join operations with indices in MIDAS and WiSS reveal similar performance characteristics (Q12, Q13, Q15, Q16). In the case without indices, the major factor of response time is depending on the overhead of sorting two relations. While MIDAS makes use of 9 buffers for sorting (8-way merge-sort), 26 buffers in WiSS are dedicated to sort area (25-way merge-sort). Since a 10,000 tuple relation occupies about 500 pages and merge-sort algorithm runs in $\lceil \log_{km} \rceil$ stages where k is the number of input buffers (ways) and m is the number of pages [Horo85], 25-way merge-sort requires 2 stages, but 8-way merge-sort runs in 3 stages. WiSS also provides a function to compare two fields in different relations in buffer pages without copying them to another temporary area. These differences resulted in the gap between the performance of join queries without indices of MIDAS and WiSS (Q9, Q10).

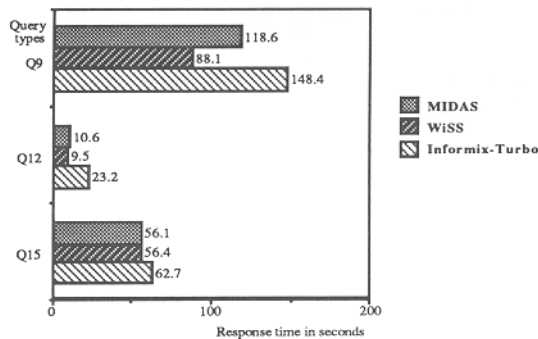


Figure 8a. Joins after a selection.

Aggregates

The queries for aggregates investigate the performance of a query optimizer and a query evaluation subsystem, instead of an access method subsystem. Only the queries to find the record whose key is minimum without index (Q20) or with a clustered index (Q23) are run in this benchmark to test the performance of comparison functions and the index search facility of MIDAS.

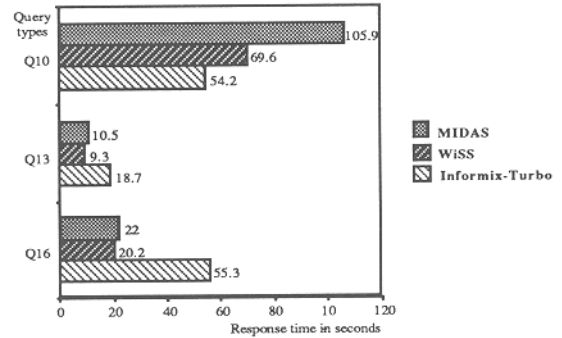


Figure 8b. Joins without selections

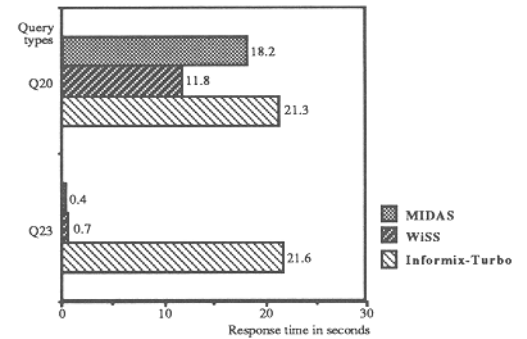


Figure 9. Aggregate queries

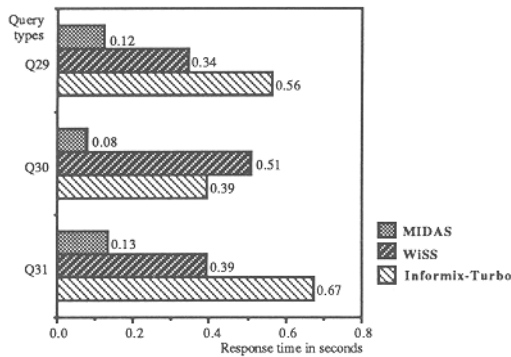
Figure 9 shows the elapsed time of aggregate queries in seconds. WiSS provides a function to compare the content of a record with a constant without copying the record into another area. The cursor with a filter in MIDAS is not applicable to processing such kind of queries because the filter of a cursor cannot be changed once the cursor is created. Using this feature, WiSS finds the minimum value with the least response time in the case of no indices (Q20). The response time of Q23 in Informix-Turbo issues a doubt that it does not optimize aggregates which appears in select-clause even though the appropriate index exists.

Updates

When these queries are executed without indices, the dominant factor of response time is not the time for update but the elapsed time to find the record to be affected. We, therefore, skips the queries without indices and run the queries with indices. Since MIDAS does not support a primary key concept such as not null and uniqueness, issued from the relational data model, the update of a key (Q31) has the same procedure of update of a nonkey (Q32). For these reasons, only following operations on a ten thousand relation with three indices are included in the benchmark: insert a tuple (Q29), delete a tuple (Q30) and update a field of a tuple (Q31).

On update queries, it is expected that the pipe, fork, query compilation and optimization overhead occupies considerable portion of response time in Informix-Turbo. We observe, however, that the overhead is so less than a second from Figure 10.

MIDAS has adopted a simple update policy for data records and index records, that is, an update of a record does not compress the page until a periodic reorganization. We analyze that the simplicity of update operations makes our system about three times faster than WiSS (Q29, Q30, Q31). Generally speaking, the overhead of the periodic reorganization caused by the simple update policy in MIDAS should



have been also measured. We also believe that the major cost of updates in real environments consists of the overhead caused by concurrency control and preparation of recovery from failures rather than updates of data and index records.

5 Concluding Remarks

MIDAS is a multiuser storage system which is intended to be used for a lower subsystem of database management systems running on UNIX. We modified B⁺-tree structure for the sake of control of concurrent operations on a B⁺-tree index. To guarantee the more concurrent accesses to the system tables such as buffers, lock table, etc., MIDAS divides each system table into several independent partitions. A MIDAS database can have more than one disk volume and a file can be spread over several volumes to store very large volume of data. We overcame the flaws of the UNIX file system efficiently by implementing an extent-based disk manager by means of the raw device interface. We adopted the two phase locking method in conjunction with multiple granularity to serialize executions of several transactions.

In order to verify design and implementation rationales which we employed, we compared the performance of MIDAS with Informix-Turbo and WiSS by use of the Wisconsin benchmark. The queries used in this benchmark have tested various access paths provided by MIDAS. The successful execution of the benchmark programs implies that MIDAS works correctly for most of access paths. From the response times we can conclude that MIDAS is a comparable storage system to existing ones such as WiSS and the lower subsystem of Informix-Turbo and that MIDAS is expected not to cause a serious performance bottleneck when it is used for a lower subsystem of database management systems.

The followings are to be done in a near future.

- Currently, MIDAS has no recovery facility. This function has been already designed and is being implementation.
- We will test the performance and functionality of MIDAS in multiuser environments. The results can lead us to tune the performance parameters and stabilize the MIDAS functionality.
- We will investigate the performance characteristics of the relationships between DBMS processes and application processes in tightly coupled multiprocessor computers by a simulation method. The function of communication management in MIDAS will be changed to comply with the simulation results.

References

- [Agra83] R. Agrawal, *Concurrency Control and Recovery in Multiprocessor Database Machines: Design and Performance Evaluation*, Ph.D. Thesis, Computer Science Dept., Univ. of Wisconsin-Madison, Madison, 1983.
- [Andr89] J. Andrade, M. Carges and K. Kovach, Building a Transaction Processing System on UNIX Systems, *Proc. UNIX Transaction Processing Workshop*, 1989, pp. 13-22.
- [Astr76] M. Astrahan et al., System R: Relational Approach to Database Management, *ACM Trans. on Database Systems*, Vol. 1, No. 2, 1976, pp. 97-137.
- [Baye77] R. Bayer and M. Schkolnick, Concurrency of Operations on B-trees, *Acta Informatica*, Vol. 9, 1977, pp. 1-22.
- [Bern87] P. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.
- [Bitt83] D. Bitton, D. DeWitt and C. Turbyfill, Benchmarking Database Systems: A Systematic Approach, *Proc. 9th Conf. on Very Large Data Bases*, 1983, pp. 8-19.
- [Bitt88] D. Bitton and C. Turbyfill, A Retrospective on the Wisconsin Benchmark, in *Readings in Database Systems* (edited by M. Stonebraker), Morgan Kaufmann Publishers, Inc., 1988, pp. 280-299.
- [Blas79] M. Blasgen, J. Gray, M. Mitoma and T. Price, The Convoy Phenomenon, *ACM Operating Review*, Vol. 13, No. 2, 1979, pp. 20-25.
- [Care84] M. Carey and M. Stonebraker, The performance of Concurrency Control Algorithms for Database Management Systems, *Proc. 10th Conf. on Very Large Data Bases*, 1984, pp. 107-118.
- [Chou85a] H-T. Chou, D. DeWitt, R. Katz and A. Klug, Design and Implementation of the Wisconsin Storage System, *Software-Practice and Experience*, Vol. 15, No. 10, 1985, pp. 943-962.
- [Chou85b] H-T. Chou and D. DeWitt, An Evaluation of Buffer Management Strategies for Relational Database Systems, *Proc. 11th Conf. on Very Large Data Bases*, 1985, pp. 127-141.
- [Come79] D. Comer, The Ubiquitous B-Tree, *ACM Computing Surveys*, Vol. 11, No. 2, 1979, pp. 121-137.
- [deJo90] W. de Jonge and A. Schijf, Concurrent Access to B-trees, *Proc. Conf. on Databases, Parallel Architectures, and Their Applications (PARBASE-90)*, 1990, pp. 312-320.
- [Fagi79] R. Fagin, J. Nievergelt, N. Pippenger and H. Strong, Extendible Hashing - A Fast Access Method for Dynamic Files, *ACM Trans. on Database Systems*, Vol. 4, No. 3, 1979, pp. 315-344.
- [Ghan89] S. Ghandeharizadeh and D. DeWitt, *Extensions to WiSS for Supporting Multiple, Concurrent Users*, unpublished manuscripts, Univ. of Wisconsin at Madison, 1989.
- [Gray78] J. Gray, Notes on Database Operating Systems, in *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, Vol. 60 (edited by Bayer-Seegmuller), Springer Verlag, 1978, pp. 393-481.
- [Hærd84] T. Hærdér and P. Peinl, Evaluating Multiple Server DBMS in General Purpose Operating System Environments, *Proc. 10th Conf. on Very Large Data Bases*, 1984, pp. 129-140.
- [Horo85] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, 1985.
- [Info87] Informix Software, Inc., *Informix-Turbo: Enhanced Database Server*, Informix Software, Inc., Menlo Park, 1987.
- [Jauh90] R. Jauhari, M. Carey and M. Livny, Priority-Hints: An Algorithm for Priority-Based Buffer Management, *Proc. 16th Conf. on Very Large Data Bases*, 1990, pp. 708-721.

- [Jian88] B. Jiang, Deadlock Detection Is Really Cheap, *ACM SIGMOD Record*, Vol. 17, No. 2, 1988, pp. 2-13.
- [Kim89] P-C. Kim et al., MUSE : A Centralized Transaction Processing System, *Proc. 3rd Pan Pacific Computer Conf.*, 1989, pp. 469-476.
- [Kim90] P-C. Kim et al., *A Benchmark of MIDAS*, Tech. Report, TM90-002, Database Lab., Computer Science Dept, KAIST, Seoul, 1990.
- [Lehm81] P. Lehman and S. Yao, Efficient Locking for Concurrent Operations on B-trees, *ACM Trans. on Database Systems*, Vol. 6, No. 4, 1981, pp. 650-670.
- [Mond85] Y. Mond and Y. Raz, Concurrency Control in B⁺-trees Databases Using Preparatory Operations, *Proc. 11th Conf. on Very Large Data Bases*, 1985, pp. 331-334.
- [Pete83] R. Peterson and J. Strickland, Log Write-Ahead Protocols and IMS/VS Logging, *Proc. 2nd ACM SIGACT-SIGMOD Sympo. on Principles of Database Systems*, 1983, pp. 216-243.
- [Sacc86] G. Sacco and M. Schkolnick, Buffer Management in Relational Database Systems, *ACM Trans. on Database Systems*, Vol. 11, No. 4, 1986, pp. 473-498.
- [Sagi85] Y. Sagiv, Concurrent Operations on B-trees with Overtaking, *Proc. 4th ACM SIGACT-SIGMOD Sympo. on Principles of Database Systems*, 1985, pp. 28-37.
- [Ston81] M. Stonebraker, Operating System Support for Database Management, *Comm. ACM*, Vol. 24, No. 7, 1981, pp. 412-418.
- [Tay85] Y. Tay, N. Goodman and R. Suri, Locking Performance in Centralized Databases, *ACM Trans. on Database Systems*, Vol. 10, No. 4, 1985, pp. 415-462.